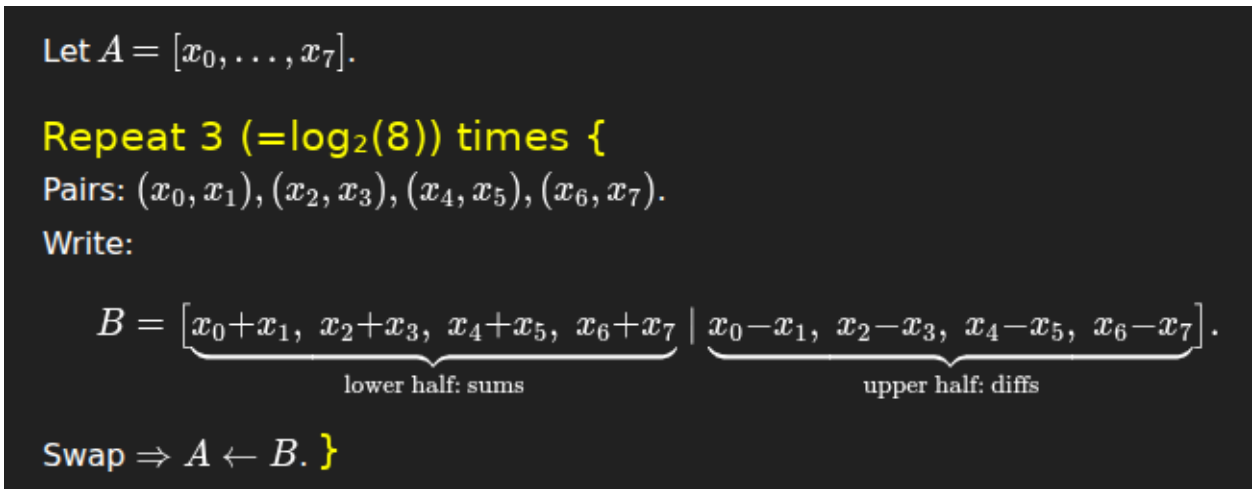
Report on the Out-of-Place Fast Walsh–Hadamard Transform

1. The Algorithm in a Nutshell

The **out-of-place FWHT** works by shuttling data between two arrays, A (input/work) and B (output/work), in a structured way:

- Step through A reading pairs of consecutive values (a_{2i},a_{2i+1})
- Compute their **sum** and place it sequentially in the **lower half** of B.
- Compute their **difference** and place it sequentially in the **upper half** of B.
- Swap roles of A and B, and repeat this process for $\log_2 N$ stages.

At the end, A (or B, depending on even or odd number of stages) contains the Walsh–Hadamard transform of the original data.


Let $A = [x_0, \ldots, x_7]$.
Repeat 3 (=log₂(8)) times {
Pairs: $(x_0, x_1), (x_2, x_3), (x_4, x_5), (x_6, x_7)$.
Write:
$B = [\underbrace{x_0+x_1,\ x_2+x_3,\ x_4+x_5,\ x_6+x_7}_{\text{lower half: sums}} \mid \underbrace{x_0-x_1,\ x_2-x_3,\ x_4-x_5,\ x_6-x_7}_{\text{upper half: diffs}}]$.
Swap $\Rightarrow A \leftarrow B$. }

Algorithm example for n=8.

This procedure is governed by a simple block matrix S of the form:

$$S = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}.$$

Example (8 by 8) S matrix

S is applied repeatedly across the array. After $\log_2 N$ stages, the composition of all these structured S-operations yields the full Hadamard matrix H□. The transpose of S can also be used.

---

2. Orthogonality and Normalization

By normalizing each S with 1/√2 every stage preserves the **Euclidean norm** of the data. This makes the transform energy-preserving and orthogonal — a key property that ensures invertibility and stability.

Because the Hadamard transform is **self-inverse** (H□²=I), the same algorithm can be used to recover the original data.

---

3. What Makes It “Out-of-Place”

The in-place FWHT overwrites data in place using butterfly operations.
The out-of-place version instead moves whole blocks of sums and differences into distinct halves of a new buffer.

This makes the data flow **simpler and more regular**: sums always pack neatly into the bottom half, differences into the top half. That regularity gives it a natural matrix representation (via repeated S-applications), and makes it easier to conceptualize than the bit-reversal-laden in-place algorithm.

---

4. Compression Angle — Beyond the Output

The out-of-place FWHT has a very appealing byproduct: it generates **structured intermediate states**.

- At each stage of the forward transform, we get arrays that mix partial sums/differences at different scales.
- At the end, we have the full Walsh–Hadamard spectrum.
- By symmetry, applying the same process again (inverse transform) takes us back to the input — but with **a fresh set of intermediate states** along the way.

So the **full set of exploitable values** for compression is:

1. The original input.
2. All forward intermediate arrays.
3. The final FWHT output.
4. All inverse intermediate arrays until the input reappears.

That's a **closed cycle** of transform states, each containing different structured mixtures of the input.

---

5. Compression Strategies

Because the FWHT is energy-preserving, **large-magnitude coefficients** are meaningful — they correspond to strong correlations with particular Walsh functions (square-wave-like basis functions).

Compression schemes could then:

- **Top-k pruning**: keep the k largest coefficients across all intermediate states, discard the rest, and reconstruct approximately.
- **Iterative max removal**: repeatedly identify the largest coefficient, remove its contribution, record its value/position, and continue.

- **Hybrid search**: allow both forward and inverse intermediates, enlarging the pool of candidate coefficients.

This approach resembles **wavelet thresholding**, but instead of multiscale wavelet filters, we're exploiting the repeated structure of the FWHT itself.

---

6. Why It's Interesting

- The out-of-place structure is **regular and easy to map to hardware**.
- Its intermediate states create a **rich hunting ground for compression** — far more than the final transform alone.
- It blends the simplicity of Walsh analysis with ideas from wavelet and sparse approximation methods.

In short: this is a transform not just to use, but to *explore*.

Code (Java):

```
public class SMatrix {

    public static void wht(float[] a) {

        float[] b=new float[a.length];

        int log2=Integer.bitCount(a.length-1); // log2 of a.length

        for(int i=0; i<log2; i++) {

            s(b,a);

            float[] t=b; // Swap arrays

            b=a;

            a=t;

        }

        if((log2&1)==1) { // if odd copy to input array (=b after swap)

            System.arraycopy(a,0,b,0,b.length);

        }

    }
```

```
public static void whtAlt(float[] a) {
    float[] b=new float[a.length];
    int log2=Integer.bitCount(a.length-1);
    for(int i=0; i<log2; i++) {
        sTranspose(b,a);
        float[] t=b; // Swap arrays
        b=a;
        a=t;
    }
    if((log2&1)==1) { // if odd copy to input array (=b after swap)
        System.arraycopy(a,0,b,0,b.length);
    }
}
public static void s(float[] b,float[] a) {
    int m=b.length/2;
    for(int i=0; i<m; i++) {
        float x=a[2*i]; // step pair-wise through a
        float y=a[2*i+1];
        b[i]=x+y;                    // sum in the lower part
        b[i+m]=x-y;   // difference in the upper part
    }
}

public static void sTranspose(float[] b,float[] a) {
    int m=b.length/2;
    for(int i=0; i<m; i++) {
        float x=a[i]; // upper part
        float y=a[i+m];// lower part
```

```
        b[2*i]=x+y;    // sum part of pair
        b[2*i+1]=x-y;// difference part of pair
    }
}



public static void main(String[] args) {
    System.out.println();
    System.out.println("N=8");
    float[] x= {1f,2f,3f,4f,5f,6f,7f,8f};
    System.out.println();
    System.out.println("FWHT & inverse using S matrix.");
    wht(x);
    System.out.println(java.util.Arrays.toString(x));
    wht(x);
    System.out.println(java.util.Arrays.toString(x));
    x=new float[] {1f,2f,3f,4f,5f,6f,7f,8f};
    System.out.println();
    System.out.println("FWHT & inverse using S Transpose matrix.");
    whtAlt(x);
    System.out.println(java.util.Arrays.toString(x));
    whtAlt(x);
    System.out.println(java.util.Arrays.toString(x));
    System.out.println();
    System.out.println("N=16");
    x=new float[] {1f,2f,3f,4f,5f,6f,7f,8f,9f,10f,11f,12f,13f,14f,15f,16f};
    System.out.println();
    System.out.println("FWHT & inverse using S matrix.");
    wht(x);
```

```
        System.out.println(java.util.Arrays.toString(x));
        wht(x);
        System.out.println(java.util.Arrays.toString(x));
        x=new float[] {1f,2f,3f,4f,5f,6f,7f,8f,9f,10f,11f,12f,13f,14f,15f,16f};
        System.out.println();
        System.out.println("FWHT & inverse using S Transpose matrix.");
        whtAlt(x);
        System.out.println(java.util.Arrays.toString(x));
        whtAlt(x);
        System.out.println(java.util.Arrays.toString(x));
    }
}
```